Adaptive Streaming and Rendering of Large Terrains using Strip Masks

Joachim Pouderoux, Jean-Eudes Marvie

Adaptive Streaming and Rendering of Large Terrains using Strip Masks

Joachim Pouderoux* Jean-Eudes Marvie†
IPARLA Project (LaBRI - INRIA Futurs)
University of Bordeaux, France

Abstract

Terrain rendering is an important factor in the rendering of virtual scenes. If they are large and detailed, digital terrains can represent a huge amount of data and therefore of graphical primitives to render in real-time. In this paper we present an efficient technique for out-of-core rendering of pseudo-infinite terrains. The full terrain height field is divided into regular tiles which are streamed and managed adaptively. Each visible tile is then rendered using a precomputed triangle strip patch selected in an adaptive way according to an importance metric. Thanks to these two levels of adaptivity, our approach can be seen as a cross-platform technique to render terrains on any kind of devices (from slow handheld to powerful desktop PC) by only exploiting the device capacity to draw as much triangles as possible for a target frame rate and memory space.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism;

Keywords: Terrain rendering, level of detail, streaming, adaptive rendering, handhelds.

1 Introduction

Terrains are involved in a lot of modern computer based applications such as geographic information systems (GIS), video games, flight simulators, etc. For the sake of realism it is necessary to use large and accurate digital elevation models (DEM). DEM are generally acquired using dual satellite images or synthetic aperture radar. Those models are available through different providers like the United States Geological Survey (USGS) or the Institut Géographique National (IGN) in France. Older terrain models can be reconstructed using contour lines extracted from scanned topographic maps [Pouderoux et al. 2004].

Rendering accurate terrains implies the manipulation of very large data sets that may contain billions of samples (e.g. triangles, points, voxels, etc.). Such a complexity introduces two main limitations ; it might not be possible to store the entire data set in memory (RAM) and/or to perform its rendering in real time on a given device. Most of the existing approaches thus propose a simplification of a triangulated model that represents the terrain surface. As we will see in section 2, some solutions entirely rely on CPU whereas others use both CPU and GPU (sometimes using programs). Among these solutions, some require the full data set to fit in memory whereas others provide out-of-core or streaming techniques.

*email: pouderou@labri.fr
†email: marvie@labri.fr

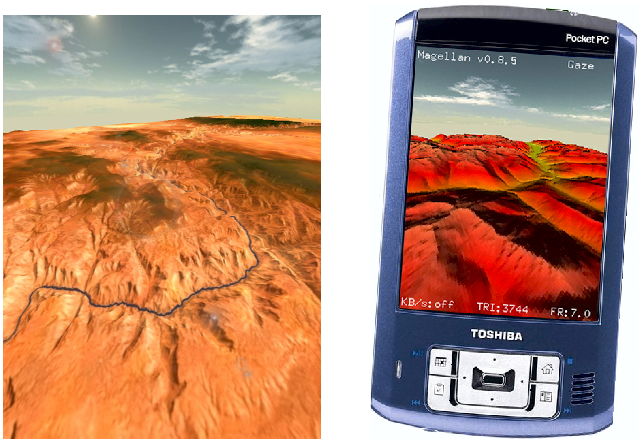

Figure 1: Streaming and adaptive rendering of large terrains. Left: adaptive rendering of the entire Grand Canyon model on a workstation at a target of 25fps, using 440K triangles. Right: adaptive rendering of the Puget Sound terrain model on a PocketPC at a target of 7fps, using 3744 triangles and streamed through an USB2.0 connection.

In our solution we want to cover a wide range of configurations (see figure 1). We thus choose to stream the terrains data in order to support networked as well as local architectures. For this purpose we use a regular tiling of the terrain in order to perform its streaming as well as memory adaptation on the client side. Targeting networked configurations implies performing interactive rendering on different types of terminals, ranging from high-end workstations to handhelds. We thus propose a multi-resolution representation (named strip masks) for adaptive rendering of each visible tile (using CPU or GPU without programs). This representation, used together with a visual importance metric and an automatic polygon budget allocator, allows to perform adaptive rendering on the client side. Our solution, thanks to its streaming and adaptive aspects, is thus widely portable and has been tested successfully on different configurations.

2 Previous work

A lot of research and engineering work has been done in the last decades in the terrain rendering domain. In this section we distinguish two families of methods. The first one brings together methods that have been designed for terrains models that fit in memory. The second family gathers the algorithms designed for the rendering of very large terrain data which cannot be completely loaded into memory (out-of-core techniques).

2.1 In memory techniques

Most of the following approaches are based on the management of triangulated irregular networks (TINs). The mesh is refined in real-time according different strategies. [Lindstrom et al. 1996] introduce a real-time smooth and continuous level of details (LOD) reduction using a mesh defined by right triangles recursively subdivided according a user-specified image quality metric. In parallel to Lindstrom, [Cohen-Or et al. 1996] propose a solution for ray tracing height fields. [Roettger et al. 1998] then proposed a geomorphing algorithm to reduce the vertex popping effect of the Lindstrom's algorithm. Hoppe introduced progressive meshes (PM) in [1996] and later described its application to terrain rendering [Hoppe 1998]. Note that, even if it presents a high CPU cost, the PM solution can be easily extended to perform streaming. In [Duchaineau et al. 1997], the authors describe their ROAMing method as a very efficient algorithm based on triangle diamonds managed with split and merge operations performed using priority queues. Even if the algorithm is widely used in games industry, its implementation is tedious according to [Blow 2000]. More recently, [Levenberg 2002] propose to reduce the CPU overhead of the previous binary-triangle-tree-based level-of-detail algorithms by manipulating aggregate triangles instead of simple triangles. Since aggregate triangles are used for more than one frame, they can be cached in the video memory and thus provide a significant acceleration.

As said in [Losasso and Hoppe 2004], previous algorithms "involve random-access memory references and immediate mode rendering". Moreover, they were designed before the spread of hardware GPU and thus present high CPU costs. Nowadays GPU are able to render millions of triangles per seconds and even more when using triangle strips. Therefore, it is now interesting to design algorithms that take advantage of these capabilities.

In a recent paper, Losasso and Hoppe [2004] apply the clipmap [Tanner et al. 1998] concept to geometry for large terrains rendering. Their GPU accelerated method is based on a set of nested regular grids centered about the viewer. Geometry continuity is guaranteed by using transition regions between two grid levels using the GPU vertex shader. They use a compression algorithm to load the full terrain model in memory. However, this still requires the full CPU power to compute vertex indices at every frame. In a more recent paper, Asirvatham and Hoppe [2005] enhanced the approach by performing nearly all computations on the GPU. Furthermore, even if the method is very efficient, it relies on shaders, which is not practicable when targeting handheld devices. Indeed, even if some recent mobile devices dispose of GPUs these are not yet programmable. Finally, keeping in mind that we target networked applications and computers of variable capacities, these solutions are not practicable when targeting streaming and also rendering of terrain models that do not fit in memory.

2.2 Out-of-core techniques

With this aim in view, some other approaches propose to perform either out-of-core rendering (local solution) or streaming (networked solution) of the models.

[Pajarola 1998] extends the restricted quadtree triangulation of Lindstrom [1996] with another vertex selection algorithm and a more intuitive triangle strip construction method. This is combined with dynamic scene management and progressive meshing to perform out-of-core rendering. More recently [Cignoni et al. 2003b; Cignoni et al. 2003a] described a technique for out-of-core management and rendering of large textured terrains named batched dynamic adaptive meshes (BDAM). BDAM is based on a pair of bintrees of small TINs that are computed and optimized off-line. The batched host-to-graphics communication model guarantees overall geometric continuity, exploits programmable GPU, a compressed out of core representation and a speculative prefetching for hiding disk latency. These solutions are still unpracticable for our objectives since they rely on low latencies between mass storage and main memory. Furthermore, these solutions also present high CPU costs.

Targeting content distribution on the Web, [Reddy et al. 1999] describe TerraVision II that is a geo-referenced VRML97 terrains viewer. A quadtree hierarchy of terrain grids is computed off-line. The approach is based on the VRML97 LOD node which induces a lot of data redundancy and no care is taken to ensure continuity between different grid levels. A more advanced solution proposed by [Aubault 2003] relies on a wavelet encoding to perform terrain streaming and multi-resolution rendering. Still, this very efficient solution requires to fetch the entire model into server's memory and to perform costly computations on it.

3 Overview

Our solution can be decomposed in two main parts. The first one consists in performing the streaming of terrain data as well as its management on the client side. This part aims at preserving the largest square area, centered on the viewpoint, that can fit in the clients memory. The second part is dedicated to the adaptive rendering of this square area. Its purpose is to render a maximum number of triangles and a highest quality of texture maps while preserving a given frame rate.

In order to perform the terrain transmission and management we used a classical tiling system [Pajarola 1998; Reddy et al. 1999; Zhao et al. 2001; Larsen and Christensen 2003]. The database is generated once by subdividing the full DEM and texture of the terrain. The geometry of each tile is then encoded into a VRML file whereas its photometry is encoded into a JPEG file or a progressive texture map format we developed. Our adaptive tiling uses an implicit data structure to perform tiles management. Encoding the tiles into separated files allows easy downloading (through a simple file transfer protocol) and management of terrain data through a simple 2D array.

The adaptive rendering of the square area is performed through the use of multi-resolution tiles. Before each frame rendering a global algorithm computes, for each tile, a visual importance (that is a percentage) deduced from its height and distance from viewpoint. This percentage is then used to share some polygon and texture map global budgets (deduced from the analysis of previous frames) among the visible tiles. The most important point, that tends to preserve CPU load and to make an intensive use of triangle strips rasterization, is the multi-resolution representation we use for each tile. A tile can be rendered through a set of coarse to fine *strip masks*. A strip mask is a precomputed triangle strip and therefore describes a known number of triangles. It is thus immediate to select a given mask, according to a given polygon budget, in order to perform the adaptive rendering of a tile. On one hand this per-tile multi-resolution scheme is less geometrically optimal than local LOD algorithms, but on the other it sends most of the CPU load to the GPU. We finally propose simple and fast solutions for geomorphing between separated levels and for cracks reduction between adjacent tiles.

The rest of this paper is organized as follow. In section 4 we present our adaptive paging algorithm. Section 5 relates our adaptive rendering solution. We finally discuss some results in section 6 before we conclude in section 7.

4 Adaptive tiling

As said before we rely on a paging system to perform the progressive fetching (or transmission) of data as well as the adaptation to main memory (the client's memory for remote sessions). With our solution the database is made of a set of files, each one containing the regular terrain elevations of a tile. A main file that contains a description of the tiles grid (tile size, number of tiles, positioning, etc.) is first fetched (or downloaded) and its information are then used to manage the adaptive tiling. The solution we propose can thus be used with a simple file transfer protocol (e.g. HTTP) if the database is located on a remote server.

The tiles management algorithm we developed aims at maintaining the largest square area (made of square bands of tiles we call *belts*) around the viewpoint. This square of tiles assures that the user can always look around at 360 degrees. Our tile management is thus quite similar to a classical paging system for terrains. However, the size of the square area is set according to the available main memory which makes it adaptive to the machine that is used for visualization. The amount of memory to be used for data storage is tuned by a user parameter. This parameter represents the percentage of available memory that can be used after the non adaptive part of the environment is fetched. We usually set this parameter to 90% in order to keep a margin for further downloadings. Knowing the amount of memory to be used, the square area is maintained by tracking the viewpoint position and by fetching/removing tiles to/from memory.

Algorithm 1 Asynchronous adaptive tiling.

```
if not currently loading then
    dc = distance to the farthest complete belt contributing to the
    complete square area
    if used memory > memory limit then
        dp = distance to the farthest (most often partial) belt not
        contributing to the complete square area
        if dp > dc then
            remove all the tiles of the partial belt from memory
        end if
    else
        ask for the fetching of the missing tiles of the belt dc + 1
    end if
else
    initialize each received tile
end if
```

Algorithm 1, that is executed before each new frame generation (see section 5), illustrates our method. The distance to a belt represents the smallest number of tiles that separates the current tile (the one that contains the viewpoint) from the belt. For instance, in figure 2a, the distance of the furthest complete belt (a belt for which all tiles are fetched) is 2. In our solution fetching is asynchronous and performed in parallel to the rendering. Fetching is always performed by belt and started only if the last requested belt has been completely fetched.

If viewpoint does not move the algorithm will fetch all the belts that can fit into memory (see figure 2a), starting from the nearest one. If the viewpoint passes over an adjacent tile the algorithm will tend to maintain a square of belts centered on this new tile (that becomes the current tile) by fetching missing tiles (see figure 2b). In this example all the memory was consumed at the step depicted by figure 2a. The algorithm will thus have to remove some far tiles in order to free some memory for the fetching of new tiles. Note that the algorithm implicitly handles the case where viewpoint jumps to a new tile that is not adjacent to the current one.

In some other cases the memory might not be saturated when the current tile changes (e.g. viewpoint moves before saturation). The remaining tiles (that do not make a complete belt) are then kept in memory (i.e. tiles are cached) for an eventual further use. Figure 2c illustrates caching. To make this screen shot we first waited saturation outside of terrain boundaries (point A) before we ran quickly on the other side of terrain boundaries (point B). We can clearly see the tiles that are cached and also the tiles that are being fetched to construct a complete belt in order to enlarge the rendered area.

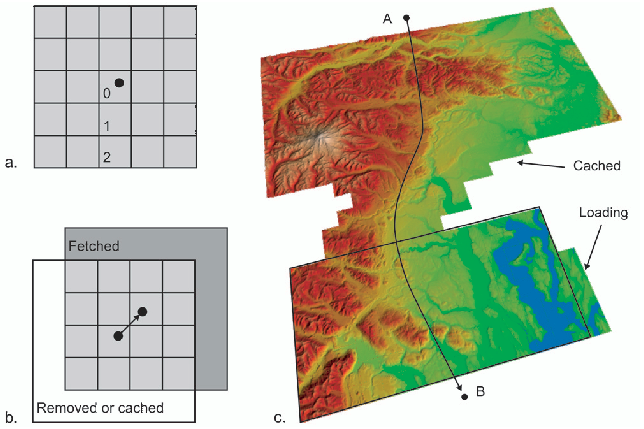

Figure 2: Tiles management and caching. a) A square area made of three belts centered on the viewpoint. b) Square area preservation on viewpoint move. c) To illustrate caching all the tiles stored in memory are rendered. Only the black rectangular area is rendered in normal usage.

Looking at the algorithm one can notice that, even if the memory budget is reached, we never remove a complete belt if it contributes to the square area. This constraints ensures the stability of the adaptive algorithm. However, the memory budget can be overspent. This is why we always keep a security margin by setting the memory percentage lower than 100%.

Finally, many solutions can be chosen if the viewpoint position is not over the terrain. A first solution could be to constrain user positions. In our case we chose to let the user fly everywhere and thus out of terrain boundaries. In order to handle this case we force the tile nearest to the viewpoint to be the current tile. This is an option that can be deactivated if, for instance, some terrains are placed side by side. Note that when approaching or leaving the boundaries, with activated option, the square area becomes a rectangular area.

5 Adaptive rendering

Recall that this step performs the rendering of every loaded tile which are visible from the current point of view (i.e. that are in the view frustum). In order to render adaptively the visible tiles, we propose a multi-resolution data structure named *strip masks* (see section 5.1). The rendering step is then performed as follows. We first compute the visual importance of each tile according to its roughness and its distance from viewpoint (see section 5.2). These visual importances are then used to share a global polygon budget (predicted to fit a given frame rate) between the visible tiles. Each partial budget (local to a tile) is finally exploited to select the strip mask to be used for the rendering of the tile (see section 5.3).

5.1 Strip mask data structure

A tile is an array of resolution $(w \times h)$ that stores elevation values of the sampled terrain area. Our implementation manages tiles of any size, but for the sake of simplicity we will only consider the specific case of tile of size $w = h = (2^n + 1)$. The memory representation of a tile is a vertex buffer that embeds the 3D vertex coordinates together with their properties like texture coordinates, normals or colors.

The multi-resolution data structure we propose is a set of strip masks of different resolutions (see figure 4). A strip mask is a regular triangulation of the tile surface at a given resolution. In practice, a mask is a triangle strip defined by an index buffer that enumerates the vertices to use related to a vertex buffer. The advantage of using triangle strips comes from the fact that modern graphic hardware (and also software graphic libraries) are optimized to render them efficiently. Moreover, as a mask is often used for more than one frame, we can also take benefit of display lists.

The main advantage of this data structure rely on the fact that all the tiles are of same resolution $(w \times h)$. A single mask set can thus be used for the multi-resolution modeling and rendering of every tile. That is to say, each index buffer of the mask set is valid for all the vertex buffers that encodes the different tiles that make up the terrain. This minimizes the computation time and specially the memory consumption. Moreover, we compute masks in a lazy way, the first time they are needed, in order to distribute the computation costs.

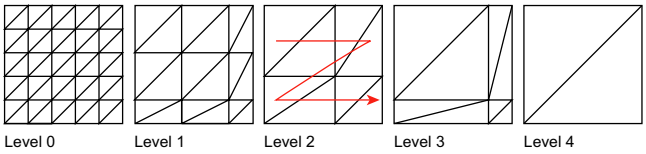

Figure 4: A simple mask set for a grid of size 6×6. The red arrow shows the order of description of the triangles in the triangle strip. Note that levels 2 and 3 both contain 8 triangles. In that case, level 2 will be chosen for a budget between 8 and 17 triangles. Level 3 will be used during a transition between level 2 and level 4 as explained in section 5.3.

The multi-resolution is generated by creating a set of coarse to fine masks. As said before, a strip mask is a regular triangulation. A mask of level l is defined by connecting only the vertices with (i, j) coordinates (within the $w \times h$ array) that are congruent modulo $l+1$. A mask set is thus made of $\max(w,h) - 1$ masks. An example of such a mask set is shown in figure 4. This approach is very different to previous ones that tend to optimize the triangulation locally according to surface properties. These previous approaches provide better decimation but consumes much more CPU resources. As the hardware is now able to render many more triangles we adopted the inverse approach. Indeed, we believe that it is preferable to render higher resolutions to produce similar qualities while preserving CPU resources to perform other kind of treatments such as complex simulations. Our data structure is conceived in this way so rendering a tile according to a polygon budget only consists in selecting the right mask. This selection is made by choosing the first level that fit the triangle budget, which is done in an $O(1)$ complexity.

5.2 Tiles visual importance

The visual importance is a percentage attributed to each visible tile according to some intrinsic and some view dependant properties. The idea is to give a higher importance and therefore more geometrical details to close and/or mountainous tiles than to far and/or flat ones. As we will see in the next section, the visual importances are then exploited to share a global polygon budget between each tile in order to select the masks that will be used for their rendering.

Algorithm 2 Computation of tiles visual importances.

```
Θ = set of visible tiles in the frustum
for each tile t in Θ do
   dist_t = distance from the camera to the center of the tile
   max_dist = max(dist_t, max_dist)
   accumulate dist_t in sum_dist
   height_t = height of the tile
   accumulate height_t in sum_height
end for
for each tile t in Θ do
   compute imp_t using equation (1)
end for
```

Algorithm 2 illustrates how we compute the visual importance of tiles. In a first loop we compute, for each tile t, $dist_t$ that is the distance from the center of the tile to the viewpoint and $height_t$ which is the height of the tile bounding box (note that one could also use a more topographical measurement like the terrain ruggedness index (TRI) developed by Riley[Riley et al. 1999]). During this loop we also store max_dist, the maximum $dist_t$ of all the visible tiles which have been processed and we accumulate $dist_t$ and $height_t$ in sum_dist and sum_height respectively. Finally, in a second loop, each tile importance imp_t is calculated as a weighted sum of the normalized $dist_t$ and $height_t$ values as follows, with $\alpha + \beta = 1$.

$$imp_t = \alpha \times \frac{max\_dist - dist_t}{sum\_dist} + \beta \times \frac{height_t}{sum\_height} \qquad (1)$$

The weights α and β are chosen empirically to accentuate or minimize the importance of tiles distance and height. To maintain a tradeoff between the distance and the height of tiles we usually use $\alpha = \beta = 0.5$. Note that the obtained visual importance values imp_t are normalized in a way that $\sum_t imp_t = 1$.

Figure 3 illustrates some visual importances obtained by setting different values for (α, β). It also shows that setting $\alpha = \beta = 0.5$ allows to preserve an accurate horizon while providing high resolutions to near tiles.

5.3 Mask selection and rendering

Once the normalized importance values imp_t are computed, each tile is registered, together with its visual importance, into a rendering-table of the Magellan [Marvie 2004] renderer. The renderer then uses these visual importance to share a global polygon budget among the different tiles. This global budget is deduced, by the renderer, from the analysis of previous frames in order to maintain a target frame rate. Budget sharing is performed by a greedy algorithm with privilege to tiles of highest priority. Basically, if the global budget for the current frame is τ triangles, the tile t will receive a drawing budget of $imp_t \times \tau$ triangles. Each time a drawing budget is allocated, the concerned tile selects the mask that present the highest amount of triangle that is lower or equal to its triangle budget. The tile then returns the amount of triangles that are not used for its rendering. This rest is re-added (by the renderer) to the global budget that is used for next tiles (of lower importance). For more details about this budget allocator see [Marvie 2004, chapter 5].

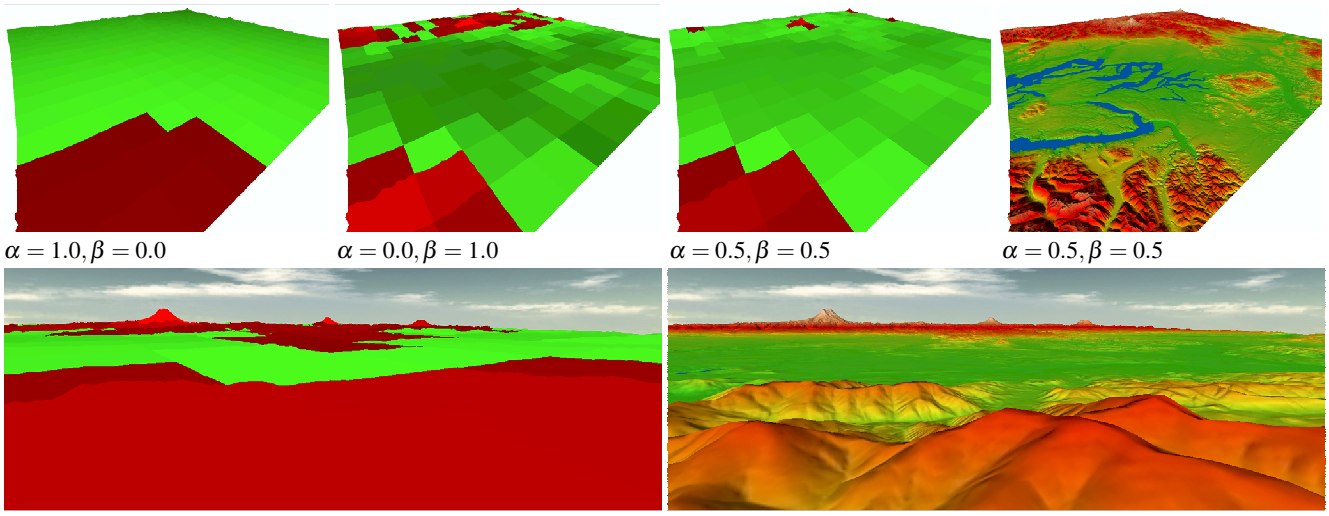

Figure 3: Visual importances on the Puget Sound model. Model elevations are exaggerated in order to see the relief better. Each tile is colored using the following color scale: red is more important than green, and light than dark. Top: from left to right, pictures show the importances using distance only, height only and both ($\alpha = \beta = 0.5$). The last image shows the textured terrain. Bottom: The left picture depicts the visual importances using $\alpha = \beta = 0.5$ which is a good tradeoff. The right picture shows the texture-mapped result. Note how the far mountains are preserved so the horizon is meaningful.

After all the tiles budgets are allocated, the renderer starts the rendering of each tile. If the selected level is the same as the one of previous frame the tile uses the display list that is already compiled. Otherwise it compiles and renders a new display list with the newly selected strip mask. Since our representation does not allow a continuous displacement of each vertex it cannot provide geomorphing. Nevertheless, we tested a geomorphing like solution which consists in rendering each intermediate level when switching from level k to level $k+l$, with $|\, l \,| > 1$. In this case we recompile a new display list only when the level $k+l$ is reached. This solution is quite simple and fast. However, it introduces several other artifacts when working on low resolution tiles and still consumes lot of AGP bandwidth, and therefore some CPU load, to perform the rendering of successive levels. Consequently, we prefer to perform direct switches of separated levels which introduce fewer visual artifacts and better performances.

Each tile can be mapped with a 2D texture. In our implementation, textures are managed as classical VRML97 textures extended with a progressive file format described in [Marvie and Bouatouch 2003]. This file format encodes mipmap levels of a texture and allows a progressive and adaptive transfer of them. This multi-resolution representation is also used during the rendering in order to optimize the GRAM occupation as well as the AGP transfers. For more details see [Marvie and Bouatouch 2003]. The important point with this solution, that is a plug-in of the Magellan renderer, is that it also takes benefits from the visual importance to update the mipmap levels (for transmission and rendering). The visual importance we compute for each tile is thus used to optimize the geometry together with the quality of texture maps.

Finally, to give a more realistic effect, it is interesting to illuminate the terrain using its normals. When light conditions are supposed to be constant the most efficient strategy is to preprocess the illumination of the terrain and to store it into the texture map. In other cases, per-vertex normals have to be recomputed into the tile's vertex buffer at each mask level update. In order to save CPU we only compute per-vertex normals once at the vertex buffer initialization and for the finest mask level only. Even if this solution is not perfect, it only introduces few artifacts when switching masks of low resolution and preserves CPU resources.

5.4 Cracks and surface continuity

When mask levels are too different between two adjacent tiles, T-vertices becomes visible and gaps appear on tiles boundaries which create a very unpleasant visual effect (see figure 5a). Classical approaches [Larsen and Christensen 2003; Losasso and Hoppe 2004] consist in modifying the geometry of the tile's border by introducing new vertices and edges. Such techniques are not compatible with our premises based on pre-computed triangle strips to ensure low CPU computation and autonomous tiles which means that a tile doesn't know the current mask level of its neighbors. Another classical method called *filleting*, introduced by Sun [1] and also implemented in the NASA's World Wind remarkable earth viewer [2] is to add a band of vertical triangles around the edges of each tile. This band is stretched down to the lowest terrain elevation. Each side of the band is textured by stretching the corresponding line/column of texels. This scheme is fast but quite disgraceful to see for the user, specially if a neighbor tile has not been loaded yet.

We propose another method which consists in drawing a planar shadow under each tile (see figure 5c). The tile shadow is a rectangular polygon made of 2 triangles drawn under each tile and texture-mapped with the same texture that the corresponding tile. The shadow position is computed as a projection of the corners of the tile regarding the current viewpoint as a classical planar shadow. For more details on planar shadows, reader can refers to [Akenine-Moller and Haines 2002, p. 250–254]. Even if this solution is not perfect and fails in some particular cases (eg. for low angles) it is fast, simple to implement and gives satisfying results most of time (see figure 5b).

[1] http://java.sun.com/products/jfc/tsc/articles/jcanyon/
[2] http://worldwind.arc.nasa.gov

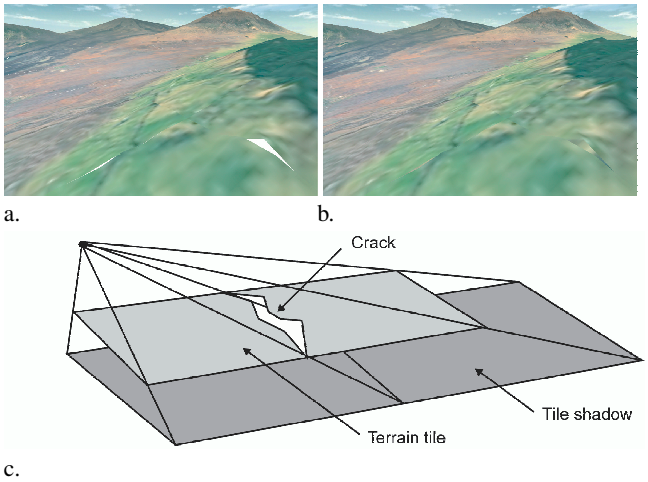

Figure 5: Crack artifacts. a) Cracks appear on tiles borders when adjacent tiles have different levels. b) Using an underlying mapped shadow plane, crack effects are attenuated. c) Two textured shadows casted by the viewpoint and rendered to fill the cracks.

6 Results

We now present some experimental results obtained with our technique. In order to prove its adaptive and streaming capabilities we performed some experimentations on different devices (handheld and desktop PC) and terrain models.

6.1 Grand Canyon

We first propose an experimentation on the model of the Grand Canyon in Arizona, USA, Initially used as an experimental model in [Hoppe 1998], the data [3] , made of a DEM and a satellite imagery, were obtained by the USGS and processed by Chad McCabe of the Microsoft Geography Product Unit. The model is a 4097×2049 grid with an inter-pixel spacing of 60 meters and a resolution of 10 meters for the elevations.

This model completely fit in the memory of the Pentium 4 (2.5GHz, 1GB of RAM, Quadro FX 500 128MB, AGP 8x) we used for the test. The terrain data and the associated texture map are subdivided into tiles of size (128×128). The resulting database is encoded using 561 zipped VRML97 files and 561 JPEG files. It occupies 26.2MB on the disk. The average size of tile files (using the VRML97 binary zipped format proposed in Magellan) is equal to 50KB and the one of the JPEG files is equal to 15KB.

For this test we used a local access to the disk and set the target frame rate equal to 25. We performed a flyover during which we recorded a set of measures that are presented by figure 6. The flyover is made of the four following time ranges:

$[0s, 13s]$ The viewpoint is placed in a corner of the terrain, looking at its integrality and we wait for the data set to be entirely fetched. We can see on the bottom plots that downloadings are distributed over the $13s$ of convergence. We can also see the amount of loaded tiles getting higher until all the tiles are loaded. Since the viewpoint looks at the entire terrain the amount of rendered tiles follows the amount of loaded tiles. The top plots show that the frame rate converges quickly to the target frame rate as well as the adaptation of

[3] http://www.cc.gatech.edu/projects/large_models

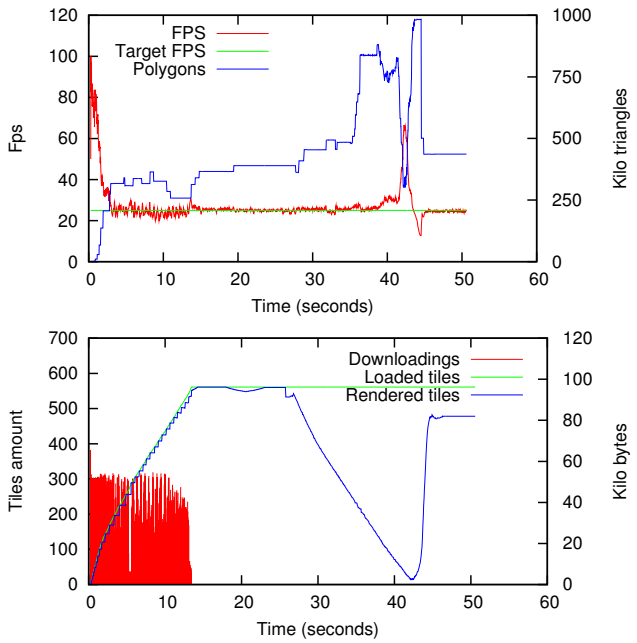

Figure 6: Performance measures for the Grand Canyon flyover. Top: evolution over time of the target frame rate, the obtained frame rate (FPS) and the amount of triangles used for each frame rendering. Bottom: evolution over time of the downloadings (expressed in Kilo bytes), the number of loaded tiles and the number of rendered tiles.

number of rendered triangles. The fluctuations of the frame rate, after this one has converged, are due to the parallelism with the thread that decodes the zipped and JPEG files and initializes the vertex arrays. Nevertheless, we can see that the frame rate always tends to fit the required frame rate.

$[13s, 19s]$ We waited a few seconds before starting the flyover. We can see that the frame rate is smoother, which is due to the fact that no more downloading is performed. We can also notice that it provides more CPU resources so the number of used triangles increases (around 100K more triangles) quickly just after the downloadings stops.

$[19s, 42s]$ We then start the flyover and cross the terrain in its length. After 7 seconds (where we fly slowly) the amount of rendered tiles decreases (thanks to frustum culling) to a very low value because we reach the opposite border of the terrain. Top plots show that the amount of rendered triangles increases massively during this period and we can clearly see an inflexion point at 35 seconds. Before 35 seconds the augmentation of the triangles amount is due to the fact that less tiles are processed which consumes less CPU resources during the scene graph traversal (frustum culling, importance calculus and budgets allocation). After 35 seconds the massive rise comes from the fact that all the compiled display lists stay in the GRAM so no more AGP transfer is performed apart from those of masks updates. Indeed, the inflexion point appears around 235 rendered tiles. Each tile requires $128 \times 128 \times 4 \times 5 = 512$KB for the encoding of its vertex array (texture coordinates, normals and vertices) and $128 \times 128 \times 3 = 48$KB for its texture map. Therefore, the amount of occupied GRAM is $(48 + 512) \times 235 = 128.51$MB that nearly corresponds to the 128MB of GRAM of which the graphics hardware disposes. Finally, near 42 seconds, the frame rate goes higher than 25fps which is due to the fact that border tiles contain a

number of triangles lower than the allocated budget.

$[42s, 50s]$ We finally do a u-turn followed by an elevation in order to see a major part of the terrain from a high point of view. Top plots show that the frame rate goes down to 12fps during a second (which is the duration of the frame rate smoother we use to filter its signal) before the system reacts to converge quickly to the target frame rate. This massive slow down is due to the global budget estimator of the Magellan renderer that computed a too important budget when the viewpoint was on the border of the terrain just before the u-turn.

6.2 Puget Sound

Our second experimentation is performed on a data set that model an area sampled near the Puget Sound region in the Washington State, USA. The data set [3] processed by [Lindstrom and Pascucci 2001] was obtained from the USGS by the way of the University of Washington. The model is made of 16385×16385 samples at 10m spacing with 16bits elevation values at 0.1m resolution. The DEM is available with an artificial texture map computed from the terrain elevations.

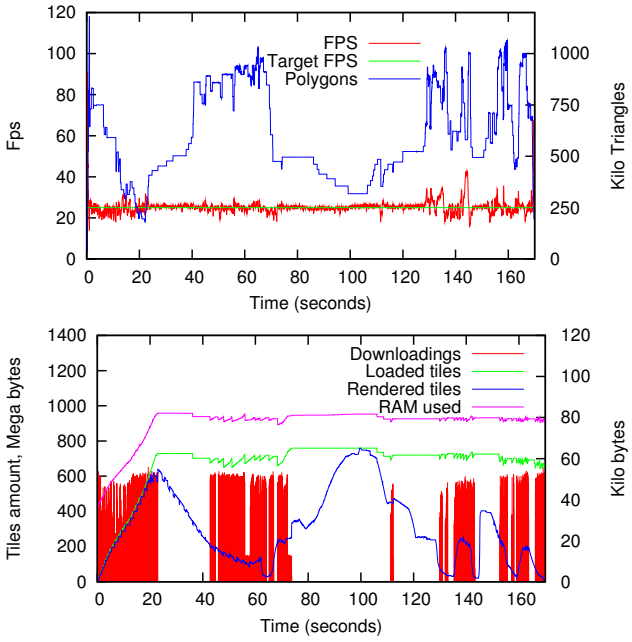

Figure 7: Performance measures for the Puget Sound flyover. Top: evolution over time of the target frame rate, the obtained frame rate (FPS) and the amount of triangles used for each frame rendering. Bottom: evolution over time of the downloadings (expressed in Kilo bytes), the number of loaded tiles and the number of rendered tiles. We also added the amount of used main memory that is correlated with the amount of tiles.

This time the model does not completely fit in the memory of the Pentium 4 we used for the test. As for the previous model, the terrain data and the associated texture map are subdivided into tiles of size (128×128). The resulting database is encoded using 8192 files (VRML97 and JPEG ones). It occupies 60MB on the disk.

For this test we used the same parameters and the same configuration as for the previous test. Figure 7 presents the recorded measures that were performed during a flyover "roaming" the entire terrain, going sometimes on its boundaries and sometimes high over it in order to get an overview. Looking at the top plots we can observe variations similar to the ones of previous test. In the bottom plots we added the amount of memory used during the flyover. We can clearly see that this plot is directly correlated with the number of loaded tiles. The results also show that we always keep a good interactivity even when streaming new data.

6.3 Puget Sound on PocketPC

In order to further validate our solution we performed a visualization of the Puget Sound model on a PocketPC Toshiba e800 (400MHz, Software OpenGL|ES implementation, display 320x240, see figure 1) connected to a server PC (the one used for the previous tests) through an USB2.0 connection. The flyover goes from one corner of the terrain to the opposite one, following the diagonal.

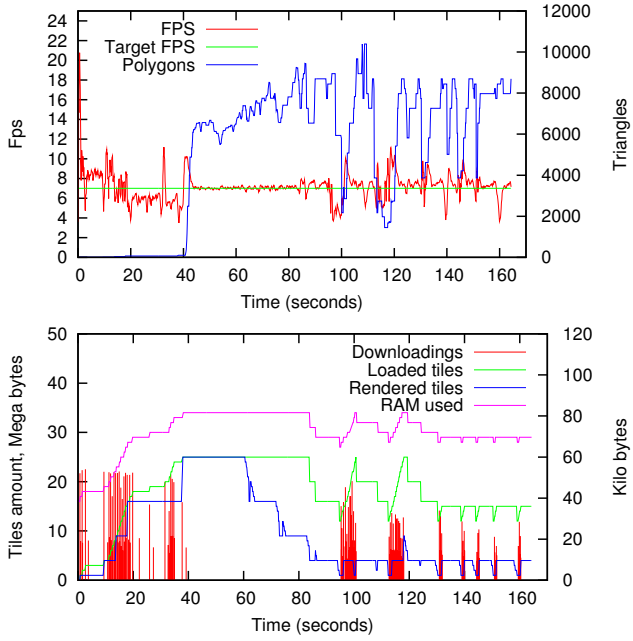

Figure 8: Performance measures for the Puget Sound flyover on a PocketPC. Measures are similar to those that are depicted by figure 7.

Figure 8 shows the measures that were performed with a target frame rate set to 7fps and 20MB of free memory. As we can see on the top plots the system adapts quite well to fit the target frame rate. However, we can notice some fluctuations when downloading in parallel (see bottom plots). These fluctuations are due to the fact that all the treatments (decompression, rendering, etc.) are performed by the CPU. It is thus extremely hard to obtain a fine parallelism. We can also see that we reach a maximum of 10000 triangles per frame, when not streaming, which is quite good if we recall that there is no GPU on the PocketPC and that the CPU only performs calculus using integer arithmetic. Indeed, in our implementation, floating points are emulated by the software and we do not even use fixed point values.

7 Conclusion

In this paper we have presented a solution that allows the streaming and the real-time rendering of large textured digital terrains. When most classical approaches make some computation to refine the model exactly where it is needed, ours favors saving CPU and memory consumption by transferring the load on the 3D graphic device. Around a tiling algorithm and a per tile multi-resolution data structure, we designed an adaptive technique with regards to the device capacities. On the one hand, the dynamic tiling management based on a memory adaptation allows a progressive downloading of data (geometry and texture maps). This allows the user to navigate immediately in the virtual environment. On the other hand, tiles are rendered efficiently using a set of masks which are precomputed triangle strips indices. Resolutions are chosen according to global and local tile properties and to the graphic hardware capabilities in order to guarantee a given frame rate. The presented results demonstrate the robustness of the adaptation.

Future work will focus on the extension of our scheme in order to use a multi-resolution data structure for the progressive and adaptive transmission of each tile as well. In that way, tiles levels could be fetched only when needed. Moreover, this will allow a faster fetching of visible tiles and offer the possibility to load farther tiles at their lowest resolution. Another investigation will be lead to avoid crack artifacts in a better way. We think this problem could be solved by adding a new mask stack for transition regions.

References

AKENINE-MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*, 2nd ed. A.K. Peters Ltd.

ASIRVATHAM, A., AND HOPPE, H. 2005. *GPU Gems II*. Addison-Wesley, ch. Terrain rendering using GPU-based geometry clipmaps, 27–44.

AUBAULT, O. 2003. *Visualisation Interactive de Scènes Vastes et Complexes à travers un Réseau*. PhD thesis, France Télécom Recherche et Développement.

BLOW, J., 2000. Terrain rendering research for games. Course on Games Research: The Science of Interactive Entertainment at SIGGRAPH.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3 (September), 505–514.

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. Interactive out-of-core visualization of very large landscapes on commodity graphics platforms. In *International Conference on Virtual Storytelling*, O. Balet, G. Subsol, and P. Torguet, Eds., vol. 2897 of *Lecture Notes in Computer Science*. November, 21–29.

COHEN-OR, D., RICH, E., LERNER, U., AND SHENKAR, V. 1996. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (sep), 255–265.

DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. ROAMing terrain: Real-time Optimally Adapting Meshes. In *Proceedings of the conference on Visualization '97*, ACM Press, 81–88.

HOPPE, H. 1996. Progressive meshes. *Proceedings of SIGGRAPH 96*, 99–108.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98*, 35–42.

LARSEN, B. S., AND CHRISTENSEN, N. J. 2003. Real-time terrain rendering using smooth hardware optimized level of detail. In *WSCG*.

LEVENBERG, J. 2002. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, 259–266.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, IEEE Computer Society, 363–371.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HUGHES, L. F., FAUST, N., AND TURNER, G. 1996. Real-time, continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH 96*, ACM SIGGRAPH / Addison Wesley, Computer Graphics Proceedings, Annual Conference Series, 109–118.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. 23*, 3, 769–776.

MARVIE, J.-E., AND BOUATOUCH, K. 2003. Remote rendering of massively textured 3D scenes through progressive texture maps. In *The 3rd IASTED conference on Visualisation, Imaging and Image Processing*, vol. 2, 756–761.

MARVIE, J.-E. 2004. *Visualisation Interactive d'Environnements Virtuels Complexes à travers des Réseaux et sur des Machines à Performances Variables*. PhD thesis, INSA de Rennes, France.

PAJAROLA, R. 1998. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization'98*, IEEE, 19–26.

POUDEROUX, J., GONZATO, J.-C., GUITTON, P., AND GRANIER, X., 2004. AutoMNT - a software for reconstructing 3D-terrains from scanned maps, aug. ACM SIGGRAPH 2004 Sketches and Applications.

REDDY, M., LECLERC, Y., IVERSON, L., AND BLETTER, N. 1999. TerraVision II: Visualizing massive terrain databases in VRML. *IEEE Computer Graphics and Applications 19*, 2 (mar-apr), 30–38.

RILEY, S., DEGLORIA, S., AND ELLIOT, R. 1999. A terrain ruggedness index that quantifies topographic heterogeneity. *Intermountain Journal of Sciences 5*, 23–27.

ROETTGER, S., HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Procceedings of WSCG '98*, 315–322.

TANNER, C. C., MIGDAL, C. J., AND JONES, M. T. 1998. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 151–158.

ZHAO, Y., ZHOU, J., SHI, J., AND PAN, Z. 2001. A fast algorithm for large scale terrain walkthrough. In *Proceedings of International Conference on CAD and Graphics 2001*.